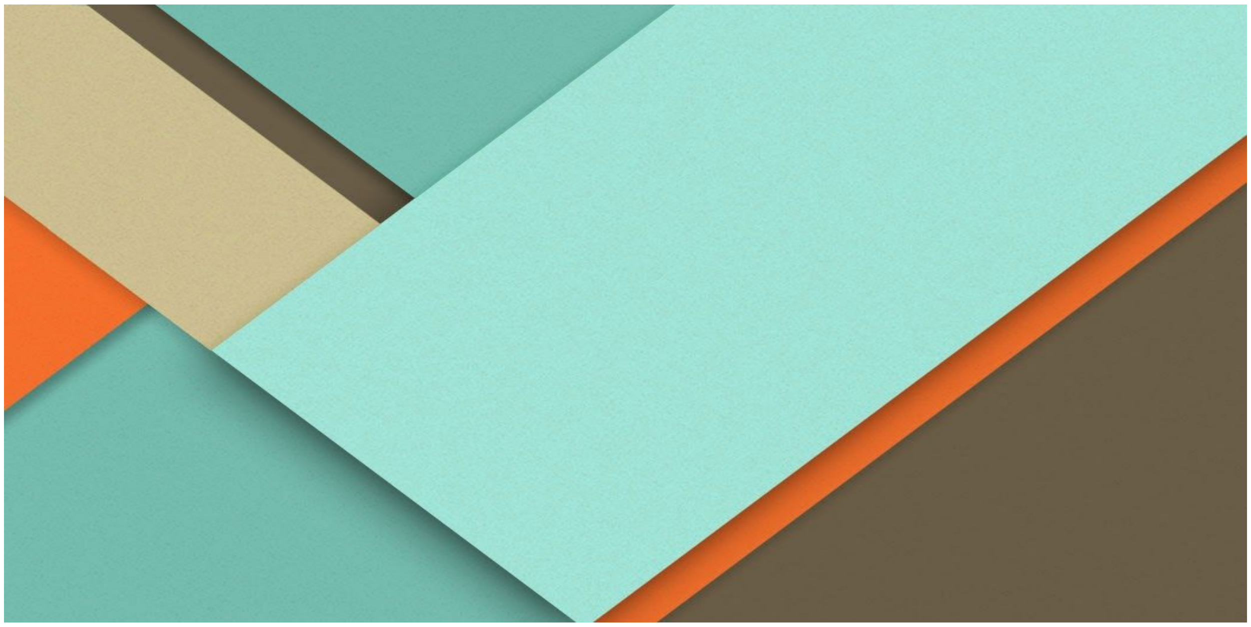


Compiler Project | Phase 1&2

15.05.2020

Ahmed Nabil (9)
Abdelrahman Alsayed Ahmed (24)
Mohamed Samy (40)
Yehia Elsayed (62)

Overview

- ### 1st Phase:

  In this project, it is required to build a compiler with its 3 first phases. In this phase, we have built the lexical analyzer generator. It is considered to be the first phase in the compiling process that translates the code into tokens which will help us in the second phase.

- ### 2nd Phase:

  This phase of the assignment aims to practice techniques for building automatic parser generator tools. The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and uses them to construct a predictive parsing table for the grammar.

Description of used Data Structures

1s Phase:

- ### Building the NFA:
- Unordered_map to save state transitions of every input
- Vector to save the states of the NFA
- Stack used in evaluation of the regular expression (transfer to postfix expression)
- ### Building the DFA:
- Vector of strings → final states of NFA, input tags, state tags[Name convention of NFA], DFA states.
- Vector of vectors of strings → DFA transitions, Partitioning.
- Map → Final states of DFA with its priority, Grouping of final states that are used in partitioning.
- Unordered_map → Final states of NFA with corresponding priorities.
- Vector of states[Implemented Class] → NFA transitions.
- ### Analyzing the Test Code:
- The same as NFA & DFA
- Vector of tokens[Implemented Class] → Tokens of the test code.

** → : contains/holds **

2nd Phase:

- **Building the Rules:**
- Map <String, Rule>: To store each non-terminal and pointer to its corresponding Rule.
- Vector<String>: To store the productions of each Rule as strings to be used later.
- **Building the Predictive Table:**
- vector<vector<string>>: To store the entries of the predictive table.
- unorderd_map<string,int>: To store each non-terminal and each terminal name and index in the predictive table.
- **Data Modeling:**
- Classes to store the required information about each rule to improve the running time and applying the OOP design concepts.

Algorithms And Techniques

1st Phase:

- **Building the NFA:**
- Use maximal munch to detect the longest valid prefix in the input as a token and then go to the next character after the end of the last token
- Use panic mode recovery when no match is found by ignoring the first character and try to match from the next one.
- **Building the DFA:**
- The process of building DFA without minimization is implemented as described in the lecture by taking the initial state of NFA, get epsilon closure of it, this state will be the initial state of DFA and applying inputs to our new initial states with getting epsilon closure of the result. If the resulting state is defined for the first time, we put it into our transitions table of DFA and repeat previous steps on it and so on.
- Minimization’s technique: Partitioning method . construct the initial partition pi of the set of states with two groups {final states , non-final states} , apply the partition method on pi , to check if each state is in its right state or need to be moved to another partition using some helping functions.the loop continues until that there is no change in the partition . for each group of states choose one state to be the representative for that group. remove the dead states and substitute each state in the transition table with its group representative.

- ## Analyzing the Test Code:
- Use maximal munch to detect the longest valid prefix in the input as a token and then go to the next character after the end of the last token
- Use panic mode recovery when no match is found by ignoring the first character and try to match from the next one.

2nd Phase:

- ## Remove Left Recursion :

  Using the algorithm of replacing each non-terminal from the last rules in current rules by its productions. And checking if it has immediate left recursion if so it will remove the left recursion according to the lectures by building the new rule for the new non-terminal and add it to the map of the rules.

- ## Left factoring :

  We have sorted vector<string> caring the productions of each rule then splitting each production by the space in a vector<vector<string>> in which each row represent a vector<string> of splitted production, then we have to check in each section of the productions beginning with the same letter for the first components of each production that is common along biggest no of productions recursively, then adding the new rule for the common factor if found.

- ## Compute first :

  Compute first of each rule recursively as there is no left recursion by looping throw every production of the rule and check for the first component of the production

  - If it is terminal: add it to the first of the rule
  - If it is nonterminal: solve it recursively and add its first to the first of the current non-terminal.

- ## Compute Follow :

  Using DFS and backtracking algorithms calculate the following set of each non-terminal by using the first sets in case of needing it. Adding $ sign to the follow set of the first non-terminal and continue with the algorithm computing the rest of the follow sets as discussed in lectures.

- ## Build the predictive table :

  We have used the algorithm of iterating through each rule with its name(Non-terminal symbol) and checking for the first terminal symbols of each production to fill its entry in the table with the production rule also looking for the epsilon in each first set and if it is found we also add the production rule in the entry of the rule and its follow terminal symbol.
  At the end if there is no epsilon in the first set of each production rule in the rule we use its entries with the follow terminal symbols as synch tokens.
  Any other undefined entries are considered as errors.

- ## Stack procedure and error handling :

We have used the general algorithm for accepting the inputs of tokens. We have used mainly stack and vectors of tokens. Also, we handled different types of errors(Terminals are not identical - Existence of “Error” in the predicative table) by taking the suitable action that is mentioned in the lecture under the headline(Panic-Mode Error Recovery).

Resultant of the stream tokens for another program

https://drive.google.com/open?id=1AixAAEd6wkaM6zUgWDsfxWPEZHpUAEqR

Test Code of the other program

https://drive.google.com/open?id=1tylFGqzhjVyv1KkN88GkA21BiL52u7ft

** Rules are the same as the project’s PDF **

** Results are provided with links due to the test code is very long to capture them as screenshots **

Predictive Table

https://drive.google.com/open?id=1BxnKjMNiGTqJ7yYdCZh82F5AWSu1pDA-

Parsing Process

https://drive.google.com/open?id=1cLRT7uVqxxUUZHDvbRpGtxxs995JbsKy

Explanation of functions of phases

After applying the lexical rules to the input then we start to use the tokens generated by the lexical analyzer as the input of the parser table token by token and use the stack procedure as we explained in the algorithms part.

Assumptions and justification

1st Phase:

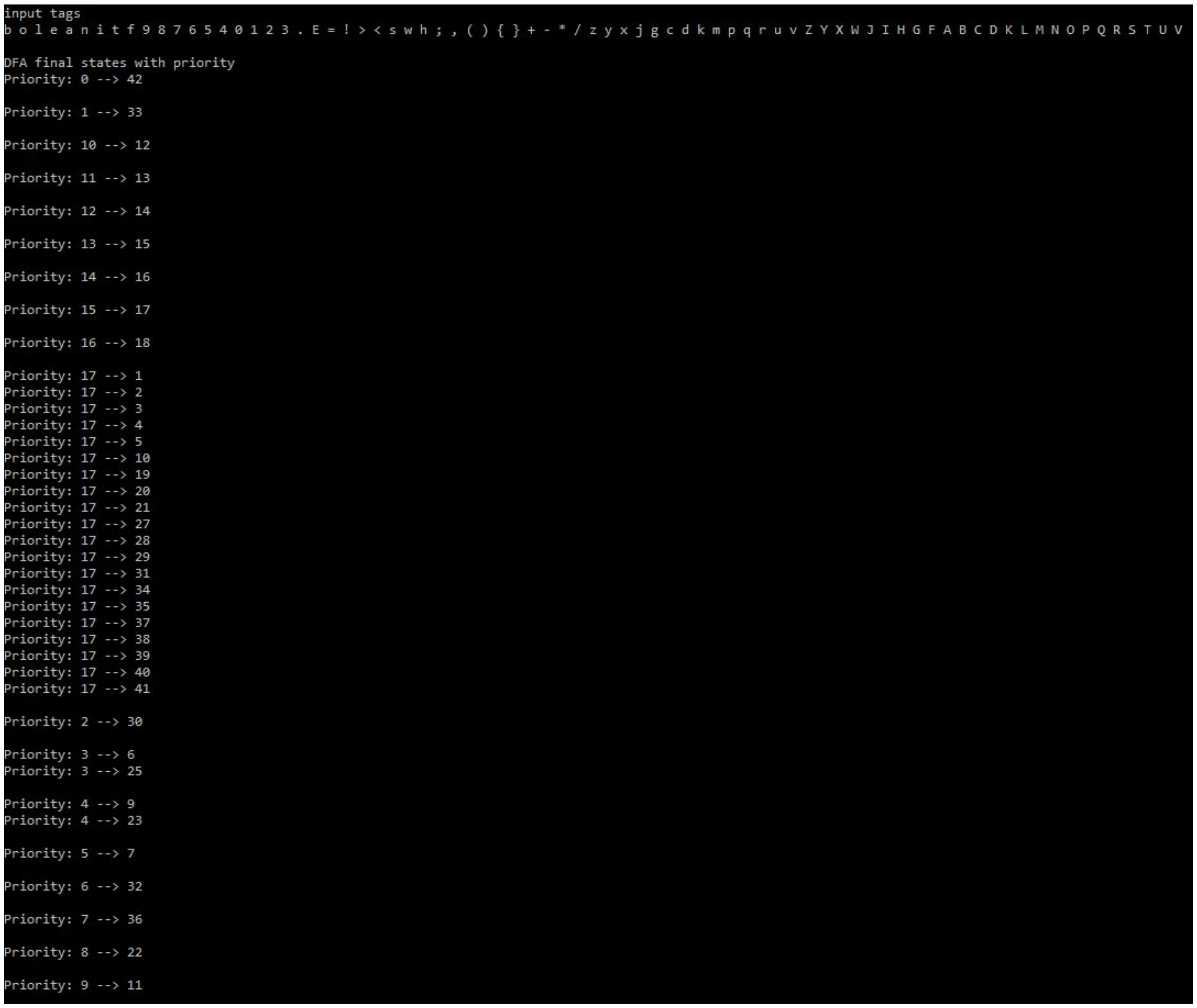
- Range operation valid only between English characters (both capital or small) or between digits from 0 to 9 And should the left character be less than the right character like b - d or 2 - 8.Any other way to describe range operation will be considered regular character
- When a rule in regular expression has an error then our code will ignore this rule.
- “Input.txt” has the input program.
- “Test.txt” has the rules.
- “Output.txt” has the output stream.

2nd Phase:

- Since we have to split the rules by the “ = “ sign and some rules have “ = “ among its production we treat the “ = “ as “ assign “ word when building the predictive table.

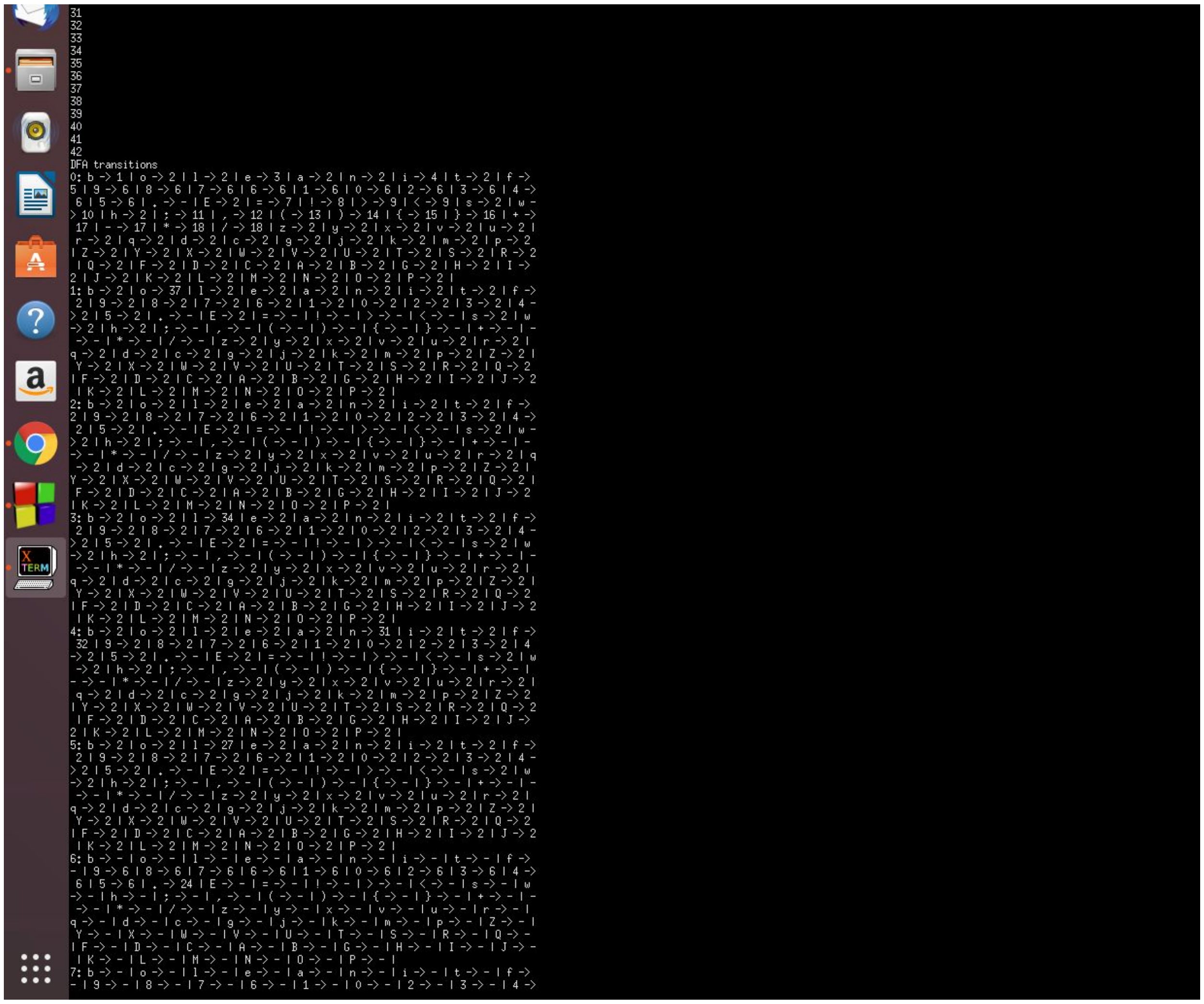
Test Case Screenshots

Resultant of the minimal DFA



```
input tags
b o l e a n i t f 9 8 7 6 5 4 0 1 2 3 . E = ! > < s w h ; , ( ) { } + - * / z y x j g c d k m p q r u v Z Y X W J I H G F A B C D K L M N O P Q R S T U V

DFA final states with priority
Priority: 0 --> 42

Priority: 1 --> 33

Priority: 10 --> 12

Priority: 11 --> 13

Priority: 12 --> 14

Priority: 13 --> 15

Priority: 14 --> 16

Priority: 15 --> 17

Priority: 16 --> 18

Priority: 17 --> 1
Priority: 17 --> 2
Priority: 17 --> 3
Priority: 17 --> 4
Priority: 17 --> 5
Priority: 17 --> 10
Priority: 17 --> 19
Priority: 17 --> 20
Priority: 17 --> 21
Priority: 17 --> 27
Priority: 17 --> 28
Priority: 17 --> 29
Priority: 17 --> 31
Priority: 17 --> 34
Priority: 17 --> 35
Priority: 17 --> 37
Priority: 17 --> 38
Priority: 17 --> 39
Priority: 17 --> 40
Priority: 17 --> 41

Priority: 2 --> 30

Priority: 3 --> 6
Priority: 3 --> 25

Priority: 4 --> 9
Priority: 4 --> 23

Priority: 5 --> 7

Priority: 6 --> 32

Priority: 7 --> 36

Priority: 8 --> 22

Priority: 9 --> 11
```

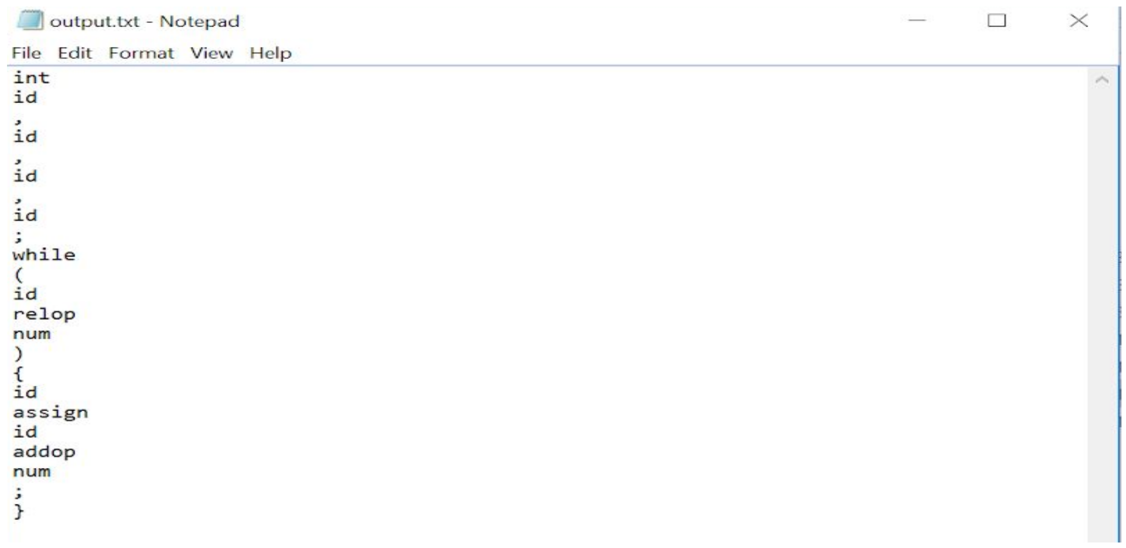
Input tags, Final states with rule priority



```
31
32
33
34
35
36
37
38
39
40
41
42
DFA transitions
0: b -> 1 | o -> 2 | l -> 2 | e -> 3 | a -> 2 | n -> 2 | i -> 4 | t -> 2 | f ->
5 | 9 -> 6 | 8 -> 6 | 7 -> 6 | 6 -> 6 | 1 -> 6 | 0 -> 6 | 2 -> 6 | 3 -> 6 | 4 ->
 6 | 5 -> 6 | . -> - | E -> 2 | = -> 7 | ! -> 8 | > -> 9 | < -> 9 | s -> 2 | w -
> 10 | h -> 2 | ; -> 11 | , -> 12 | ( -> 13 | ) -> 14 | { -> 15 | } -> 16 | + ->
 17 | - -> 17 | * -> 18 | / -> 18 | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 |
 r -> 2 | q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2
| Z -> 2 | Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2
 | Q -> 2 | F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I ->
2 | J -> 2 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
1: b -> 2 | o -> 37 | l -> 2 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
 2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 -
> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
-> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
 Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
| F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
2: b -> 2 | o -> 2 | l -> 2 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 ->
 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w -
> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
-> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 | q
 -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2 |
 F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
| K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
3: b -> 2 | o -> 2 | l -> 34 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
 2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 -
> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
-> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
 Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
| F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
4: b -> 2 | o -> 2 | l -> 2 | e -> 2 | a -> 2 | n -> 31 | i -> 2 | t -> 2 | f ->
 32 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4
-> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
 -> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - |
- -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
 q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2
| Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
 | F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J ->
2 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
5: b -> 2 | o -> 2 | l -> 27 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
 2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 -
> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
-> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
 Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
| F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
6: b -> - | o -> - | l -> - | e -> - | a -> - | n -> - | i -> - | t -> - | f ->
- | 9 -> 6 | 8 -> 6 | 7 -> 6 | 6 -> 6 | 1 -> 6 | 0 -> 6 | 2 -> 6 | 3 -> 6 | 4 ->
 6 | 5 -> 6 | . -> 24 | E -> - | = -> - | ! -> - | > -> - | < -> - | s -> - | w
-> - | h -> - | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> - | y -> - | x -> - | v -> - | u -> - | r -> - |
q -> - | d -> - | c -> - | g -> - | j -> - | k -> - | m -> - | p -> - | Z -> - |
 Y -> - | X -> - | W -> - | V -> - | U -> - | T -> - | S -> - | R -> - | Q -> -
| F -> - | D -> - | C -> - | A -> - | B -> - | G -> - | H -> - | I -> - | J -> -
 | K -> - | L -> - | M -> - | N -> - | O -> - | P -> - |
7: b -> - | o -> - | l -> - | e -> - | a -> - | n -> - | i -> - | t -> - | f ->
- | 9 -> - | 8 -> - | 7 -> - | 6 -> - | 1 -> - | 0 -> - | 2 -> - | 3 -> - | 4 ->
```

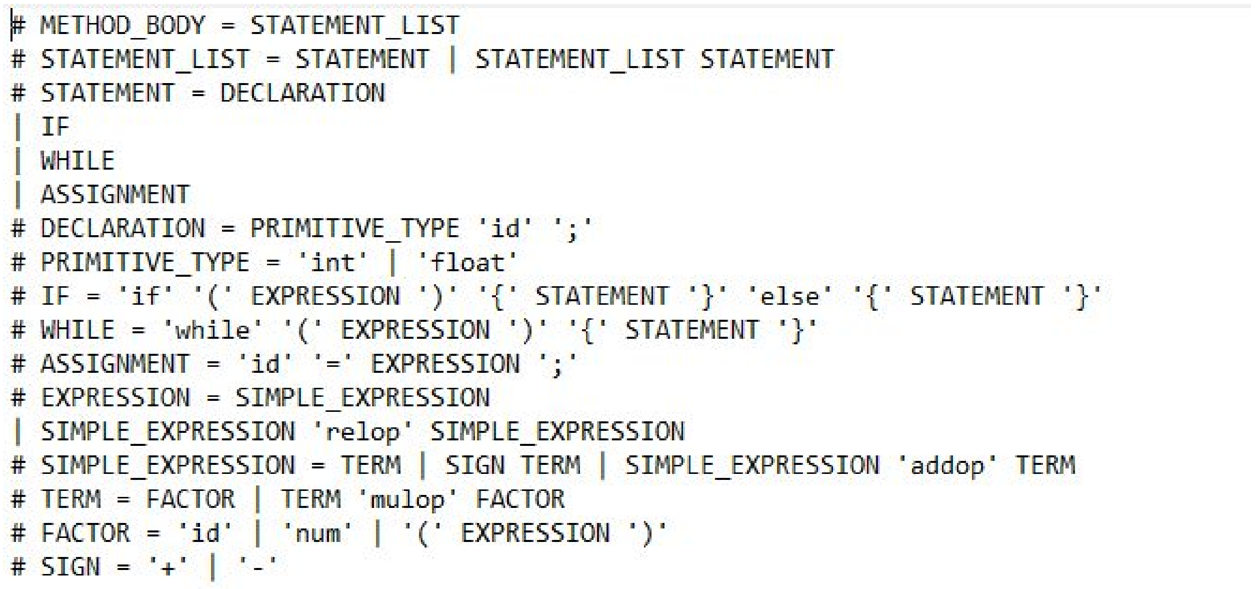
Minimal transition table of the DFA

Resultant of the stream tokens for example program



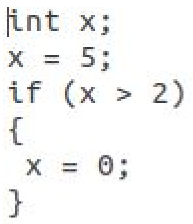
```
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```

Rules:



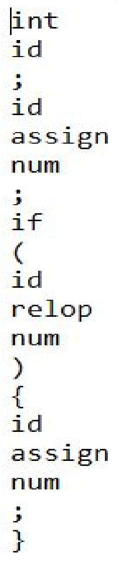
```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT = DECLARATION
| IF
| WHILE
| ASSIGNMENT
# DECLARATION = PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE = 'int' | 'float'
# IF = 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
# WHILE = 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
# ASSIGNMENT = 'id' '=' EXPRESSION ';'
# EXPRESSION = SIMPLE_EXPRESSION
| SIMPLE_EXPRESSION 'relop' SIMPLE_EXPRESSION
# SIMPLE_EXPRESSION = TERM | SIGN TERM | SIMPLE_EXPRESSION 'addop' TERM
# TERM = FACTOR | TERM 'mulop' FACTOR
# FACTOR = 'id' | 'num' | '(' EXPRESSION ')'
# SIGN = '+' | '-'
```

Input Code :



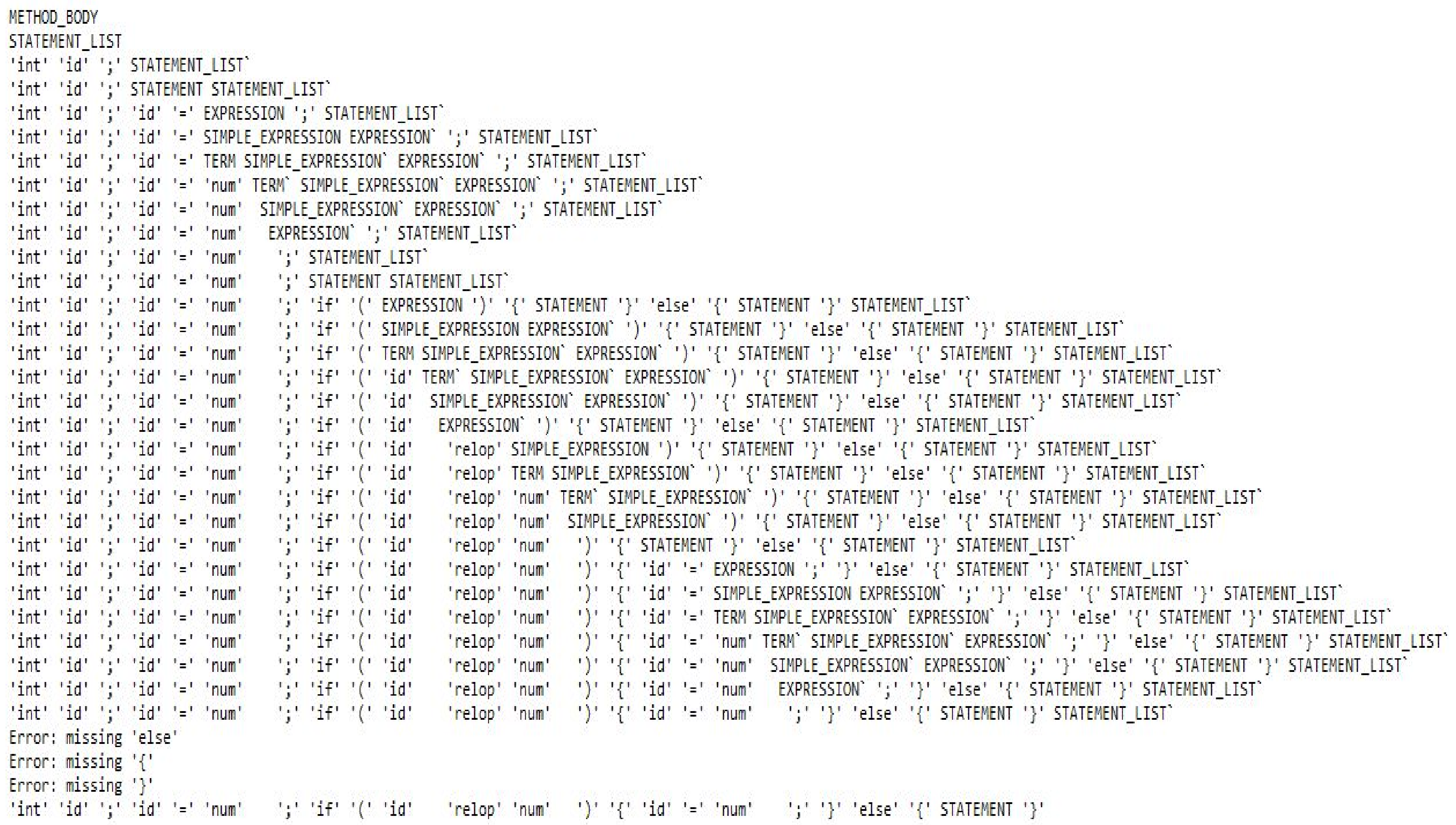
```
int x;
x = 5;
if (x > 2)
{
 x = 0;
}
```

The output of the lexical:



```
int
id
;
id
assign
num
;
if
(
id
relop
num
)
{
id
assign
num
;
}
```

The output of the Parser:



```
METHOD_BODY
STATEMENT_LIST
'int' 'id' ';' STATEMENT_LIST`
'int' 'id' ';' STATEMENT STATEMENT_LIST`
'int' 'id' ';' 'id' '=' EXPRESSION ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' SIMPLE_EXPRESSION EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' TERM SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num' TERM` SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'  SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'   EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' STATEMENT STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' SIMPLE_EXPRESSION EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' TERM SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id' TERM` SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'  SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'   EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' SIMPLE_EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' TERM SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num' TERM` SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'  SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' EXPRESSION ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' SIMPLE_EXPRESSION EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' TERM SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num' TERM` SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'  SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'   EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'    ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
Error: missing 'else'
Error: missing '{'
Error: missing '}'
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'    ';' '}' 'else' '{' STATEMENT '}'
```

/***Link to the parsing is added above ***/